OOP Notes with JAVA

1. Class:

The Blueprint (Logical Construct): A **class** in Java is a **template**, a **blueprint**, or a **user-defined data type**. It doesn't occupy memory. It just defines:

- **Fields** (data/state)
- **Methods** (behavior)
- **Constructors** (initialization logic)

Key Point: A class is a **logical construct**. It doesn't exist in memory until you **create an object**.

Example:

```
class Box {

    int length;       // field (state)
    int breadth;

    void displayArea() {  // method (behavior)
        System.out.println("Area: " + (length * breadth));
    }
}
```

Above, Box is **just a plan** — not an actual box yet.

---

2. Object: Instance of a Class (Physical Reality)

When you create an object, you're creating a **real entity in memory** based on the class.

Key Statement: A class creates a new data type; an object is an instance of that type.

Box mybox = new Box();

- Box → class name (blueprint)
- mybox → reference variable (like a remote control)
- new Box() → creates a real **object in memory**
- mybox → holds the **memory address** of that object

Objects Have Three Core Characteristics:

1. **state:** Values stored in variables (e.g., length = 5)
2. **Identity:** Unique reference to memory (e.g., where the object lives in RAM)
3. **Behaviour**: What the object can do (methods, like displayArea())

Example:

```
Box b1 = new Box();
b1.length = 10;
b1.breadth = 5;
b1.displayArea();  // Output: Area: 50
```

Here:

- **State** = length = 10, breadth = 5
- **Identity** = memory location held by b1
- **Behavior** = method displayArea()

---

3. The new Keyword – What It Really Does

Purpose:

- Allocates **memory during runtime** (dynamically)
- Returns a **reference** to the object in memory
- This reference is stored in the **reference variable**

Syntax: ClassName objectName = new ClassName();

This process:

1. JVM uses new to create object
2. Allocates memory in heap
3. Calls **constructor**
4. Returns memory reference
5. Assigns it to variable

Code:

```
Box mybox;

mybox = new Box();
```

- `Box mybox;` → declares reference (just a remote control)
- `new Box();` → creates actual object
- `mybox = ...` → mybox now points to the object

Before new: mybox points to nothing
After new: mybox holds reference to the Box object in memory

Java ensures safety: You **cannot manipulate memory addresses directly** like in C/C++.

Object References Point to Same Object

```
Box b1 = new Box();
Box b2 = b1;
```

- Both b1 and b2 refer to the **same object** in memory.
- No new object is created — only the reference is copied.
- Any changes through b2 affect the same object b1 refers to.

```
b2.length = 20;
System.out.println(b1.length);  // Output: 20
```

The Dot (.) Operator

Used to access:

- Object's fields
- Object's methods

```
Box b = new Box();
b.length = 10;    // access field
b.displayArea();  // access method
```

Formally, . is called a **separator**, but we commonly call it **dot operator**.

Constructor and new Together

```
Box b = new Box();  // calls default constructor
```

The () calls a **constructor** to initialize the object.

Parameters vs Arguments

```
int square(int i) {  // i = parameter
    return i * i;
}

square(100);  // 100 = argument
```

Parameter: Variable in method definition
Argument: Actual value passed when calling method

Java Code Example

```
class Box {
    int length = 10;
    int breadth = 5;

    void displayArea() {
        System.out.println("Area: " + (length * breadth));
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Box mybox = new Box();   // object creation
        mybox.displayArea();     // accessing method
    }
}
```

---

4. Constructor in Java

What is a Constructor?

A **constructor** is a special method that:

- Has **no return type** (not even void)
- Has the **same name** as the class
- Is **automatically called** when an object is created using new
- Is used to **initialize objects**

Syntax:
```
class Box {

    int length;
    int breadth;

    // Constructor
    Box() {
        length = 10;
        breadth = 5;
        System.out.println("Box Constructor Called");
    }
}
```

Example:
```
public class Main {

    public static void main(String[] args) {
        Box mybox = new Box(); // Constructor is called here
    }
}
```

Output: `Box Constructor Called`

Constructor Rules:

1. Constructor name = class name
2. No return type (not even void)
3. Called automatically by new
4. Can be **default** (no args) or **parameterized (with args)**

Parameterized Constructor:

```
class Box {
    int length;
```

```
    int breadth;

    Box(int l, int b) {
        length = l;
        breadth = b;
    }

    void displayArea() {
        System.out.println("Area: " + (length * breadth));
    }
}
```

Important Note: Once you define a **parameterized constructor**, Java **doesn't provide the default constructor**. You must explicitly define it if needed.

---

5. The this Keyword

What is this?: The this keyword refers to **the current object** — i.e., the object on which the method or constructor is called.

Use Cases:

1. **To Refer to Instance Variables:**

```
class Box {
    int length;

    Box(int length) {
        this.length = length;  // 'this.length' is instance variable,
'length' is parameter
    }
}
```

Without this, length = length would assign the parameter to itself (no effect).

2. **To Call Another Constructor in the Same Class:**

```
class Box {
    int length, breadth;

    Box() {
        this(10, 20);  // calls the parameterized constructor
    }

    Box(int l, int b) {
        length = l;
        breadth = b;
    }
}
```

Summary: this helps avoid naming conflicts and allows calling one constructor from another.
It always refers to the **current objectt**

6. The final Keyword

Uses of final:

- **final variable**: constant value
- **final method**: cannot be overridden
- **final class**: cannot be inherited

Final Variable (Constant):

```
class Example {
    final int FILE_OPEN = 2; // must be initialized

    void display() {
        // FILE_OPEN = 3; // Error: can't reassign final variable
        System.out.println(FILE_OPEN);
    }
}
```

Convention: use UPPERCASE for final constants.

Final with Reference Types:

```
final int[] arr = {1, 2, 3};
arr[0] = 100; //  allowed: changing object content
// arr = new int[]{4,5}; //not allowed: changing reference
```

Final prevents reference change, not object mutation.

The finalize() Method

Purpose:

- To define **cleanup actions** before an object is garbage collected
- Rarely used in modern Java due to unpredictable GC timing

Syntax:
```
class Example {

    protected void finalize() {
        System.out.println("Object is being garbage collected");
    }
}
```

Note: finalize() is deprecated in Java 9+. Better alternatives: try-with-resources, Cleaner API.

Inheritance and Constructors

Behavior: If you create a subclass object, Java first **calls the superclass constructor**, then subclass constructor.

Example: `class Base {`

```
    Base() {
        System.out.println("Base Class Constructor Called");
    }
}

class Derived extends Base {
    Derived() {
        System.out.println("Derived Class Constructor Called");
    }
}
java
Copy code
public class Main {
    public static void main(String[] args) {
        Derived d = new Derived();
    }
}
```

Output:
```
Base Class Constructor Called
Derived Class Constructor Called
```

Rules:

Case	Behavior
No constructor in subclass	Java automatically calls superclass constructor
Parameterized constructor in subclass	Still calls **default superclass constructor** unless specified
No default constructor in superclass	You **must explicitly call** parameterized constructor using super(args)

Example with Error: `class Base {`

```
    Base(int x) { } // only parameterized constructor
}

class Derived extends Base {
    Derived() {
        // Java tries to call super() here but no default constructor
exists
    }
}
```

Fix it using: `Derived() {`

```
    super(10);
}
```

---

7. Garbage Collection in Java

What is Garbage Collection? Garbage Collection (GC) **in Java is the** automatic process **of reclaiming memory by** removing unused objects **(i.e., objects that no longer have any references).**

You don't need to manually delete objects like in C/C++. Java handles it for you to prevent **memory leaks**.

Real-life Analogy: Imagine your phone's system automatically closing unused background apps to free up RAM — that's garbage collection.

Why Do We Need It? Java applications create many objects. Some become unused. GC:

- **Frees up heap memory**
- Helps **optimize performance**
- **Prevents memory leaks**

How It Works?

- The **JVM** runs a background **Garbage Collector** thread.
- GC scans the **heap** to find **unreachable objects** (objects with no live reference).
- It then **deallocates memory** used by those objects.

```
Box b1 = new Box();
b1 = null;  // now eligible for garbage collection
```

You Can *Suggest* Garbage Collection:

```
System.gc();  // Suggests JVM to run GC
```

Note: This is **just a request** — the JVM **may or may not** run GC immediately.

finalize() Method and GC

```
class Box {
    protected void finalize() {
        System.out.println("Box object is being collected");
    }
}
```

finalize() is called **just before** the object is collected — to perform **cleanup**.

Deprecated since Java 9+ (not recommended now).

---

8. @Override Annotation

What is @Override? : The @Override **annotation is used to** inform the compiler **that a method is** overriding **a method from a** superclass **or interface.**

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}
```

Why Use @Override?

1. **Ensures correctness**: If you make a typo, compiler catches it.
2. **Improves readability**
3. **Protects from errors** in overriding

Without @Override (and a typo):

```
class Dog extends Animal {
    // Typo in method name
    void snd() {  // Compiler thinks it's a new method
        System.out.println("Bark");
    }
}
```

No error, but **method isn't overridden** — logic fails silently.

With @Override, compiler will warn you.

Summary:

- Always use @Override when overriding methods
- Helps catch mistakes and improve clarity

---

9. Throwable Class in Java

What is Throwable?

Throwable is the superclass of all errors and exceptions **in Java.**
All things that can be thrown and caught **in a try-catch block must be a subclass of Throwable.**

Throwable Class Hierarchy:

```
         Throwable
         /      \
     Error     Exception
```

Exception (Recoverable Issues)

- Caused by problems **in your program**
- Can be **caught and handled**
- Example: NullPointerException, ArithmeticException, IOException

```
try {
    int x = 5 / 0;
} catch (ArithmeticException e) {
    System.out.println("Division by zero!");
}
```

Error (Unrecoverable Problems)

- Caused by **JVM or system** problems
- **Should not be caught**
- Examples: OutOfMemoryError, StackOverflowError

```
// Example: deep recursion causes stack overflow
void recurse() {
    recurse(); // never ends
}
```

Throwable's Common Methods:

Method	Description
getMessage()	Returns error message
printStackTrace()	Prints error details
toString()	Returns exception name + message

Custom Exception Example:

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}

public class Main {
    public static void main(String[] args) throws MyException {
```

```
        throw new MyException("Custom error thrown!");
    }
}
```

---

10. What is a Package in Java?

A **package** in Java is a **container** for classes, interfaces, sub-packages, and even other packages.
It helps in **organizing** your code and **avoiding name conflicts**.

Why Use Packages?

Imagine this:

You and someone else both create a class named List.
Without packages, Java won't know which List you're referring to.

But if you store your class as:

```
package myapp.datastructures;
public class List { ... }
```

…and the other class is in:

```
package java.util;
public class List { ... }
```

Then you can clearly **refer** to either using the package name:

```
myapp.datastructures.List myList = new myapp.datastructures.List();
java.util.List anotherList = new java.util.ArrayList();
```

This avoids name collisions.

Syntax to Declare a Package

```
package mypackage;

public class MyClass {
    // code
}
```

- This line **must be the first line** in your Java file.
- This tells Java: "Hey, this class belongs to the package mypackage."

How Are Packages Stored on Disk?

- Packages are stored in **directories** that match their names.
- If you write:

```
package com.adnan.math;
```

Your .java file should be stored in the folder:
```
com/adnan/math/
```

- Java is **case-sensitive**, so Math ≠ math.
- The **directory structure must exactly match** the package name

Real Example: Let's say you create this class:

```
package com.mycompany.shapes;

public class Circle {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

It should be saved at:

```
/your-project/com/mycompany/shapes/Circle.java
```

Then, in another class, you can import and use it:

```
import com.mycompany.shapes.Circle;

public class Test {
    public static void main(String[] args) {
        Circle c = new Circle();
        c.draw();
    }
}
```

Visibility and Access Control in Packages

- Only **public** classes, methods, and fields can be accessed from outside the package.
- Classes, methods, or fields without a modifier (default access) are **package-private** and can be accessed **only within the same package**.

How Java Finds Your Package Classes

1. Default – Current Directory

If your package is in a subfolder of the directory you're running the code from, Java will find it automatically.

2. CLASSPATH Environment Variable

You can set the CLASSPATH to tell Java where to look for your packages.

Example (Windows):

```
set CLASSPATH=C:\Users\Aadii\MyJavaProjects\
```

3.-classpath or -cp Option

You can also specify it directly when compiling or running:

```
javac -cp . com/mycompany/shapes/Circle.java
java -cp . Test
```

Types of Packages

1. **Built-in Packages**: Provided by Java (e.g., java.util, java.io, java.lang)
2. **User-defined Packages**: You create them as needed for modularity and code management.

Example of Built-in Package

```
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Aadii");
        System.out.println(list);
    }
}
```

Common Mistakes to Avoid

Mistake	Why it's a problem
Declaring package after import	Package declaration **must come first**
Folder name doesn't match package	Java will throw errors or not find the class
Using default access for classes/methods that need to be public	Other packages won't be able to access them
Forgetting to compile dependent packages	Compilation will fail

```
import static java.lang.Math.*;

public class Test {
    public static void main(String[] args) {
        System.out.println(sqrt(16)); // no need to write Math.sqrt()
    }
}
```

Summary

Feature	Description
What	Container for classes and interfaces
Why	Avoid name conflicts, organize code
Syntax	package mypackage; (must be topmost line)

Directory Structure	Folder names must match package names
Access	Only public members are accessible outside package

11. What is static in Java?

The static keyword in Java means **“this belongs to the class, not to any object.”**

Normally, variables and methods belong to **instances (objects)**. But when you make them static, they belong to the **class itself**.

Static Variables (Class Variables)

These are variables **shared by all objects** of a class.

```
class Counter {
    static int count = 0; // shared by all instances

    Counter() {
        count++; // count is incremented for every object created
    }

    public static void main(String[] args) {
        new Counter();
        new Counter();
        System.out.println(Counter.count); // Output: 2
    }
}
```

- Same variable for all objects
- Access using ClassName.variable
- Not tied to any object

Static Methods

A static method:

- Belongs to the **class**.
- Can be called **without creating an object**.
- **Can access only other static members** (methods or variables).

```
public class MathUtil {
    static int square(int x) {
        return x * x;
    }

    public static void main(String[] args) {
        System.out.println(MathUtil.square(5)); // Output: 25
    }
}
```

Static Methods Can’t Access Instance Variables Directly

```
class Human {
    String message = "Hello World";

    public static void display(Human h) {
        System.out.println(h.message);  // OK: using object reference
    }

    public static void main(String[] args) {
        Human h1 = new Human();
        h1.message = "Kunal's message";
        display(h1);
    }
}
```

Static Block

A static block runs **once when the class is first loaded** into memory. You use it to initialize static variables with logic (not just direct assignment).

```
class UseStatic {
    static int a = 3;
    static int b;

    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    public static void main(String[] args) {
        meth(42);
    }
}
```

Output:

```
makefile
Static block initialized.
x = 42
a = 3
b = 12
```

Why main() is Static?

The main() method is the **entry point** of any Java program. It is static because it must run **before** any object is created.

```
public static void main(String[] args)
```

- JVM directly calls it using the class name — no object exists yet.
- So it must be **static** to be called without an object.

Static Classes (Nested Static Classes)

Only **nested classes** can be static.

```
public class Outer {
    static class Inner {
        void show() {
            System.out.println("Inside static nested class");
        }
    }

    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();
        obj.show();
    }
}
```

Why only nested classes can be static?

A **static class** means "this does not need access to any instance members of the outer class."

Limitations of Static Context

- Can’t access **non-static variables** directly.
- Can’t call **non-static methods** directly.
- Can’t use this or super.
- Can use object references to access non-static stuff.

```
class A {
    int value = 10;

    public static void printValue(A obj) {
        System.out.println(obj.value);  // okay using object
    }
}
```

Static Methods Can't Be Overridden

Overriding works on **runtime polymorphism** — which is **object-based**.

Since static methods are **class-based** and resolved at **compile-time**, **you can’t override them**. You can **hide** them though:

```
class A {
    static void greet() {
        System.out.println("Hello from A");
    }
}
class B extends A {
    static void greet() {
        System.out.println("Hello from B");
    }
    public static void main(String[] args) {
```

```
        A.greet(); // Hello from A
        B.greet(); // Hello from B
    }
}
```

Static Interface Methods

In Java 8+, interfaces can have static methods too. But:

- They are **not inherited**.
- Must be called using the interface name.

```
interface MyInterface {
    static void greet() {
        System.out.println("Hello!");
    }
}
public class Demo {
    public static void main(String[] args) {
        MyInterface.greet(); // ✅
        // greet(); ❌ not inherited
    }
}
```

Summary Table:

Feature	Static	Non-Static
Belongs to	Class	Object (Instance)
Accessed via	ClassName	Object reference
Needs object?	No	Yes
Can use this/super	No	Yes
Access instance vars	Only through object	Directly
Overridable	No (only hidden)	Yes

---

12. What is Inheritance in Java?

Inheritance is the mechanism in Java by which **one class (child/subclass)** can **acquire the properties and behaviors (methods and variables) of another class (parent/superclass)**.

It models the real-world **"is-a"** relationship.

Example: A Dog **is a** Animal. So, Dog can inherit from Animal.

Syntax of Inheritance

```
class SuperClass {
    // properties & methods
}

class SubClass extends SuperClass {
    // extra properties & methods
```

```
}
```

The keyword `extends` is used for **class inheritance**.

Types of Inheritance in Java

Type	Supported in Java?	Example
Single Inheritance	Yes	class B extends A
Multilevel Inheritance	Yes	C extends B, B extends A
Hierarchical Inheritance	Yes	B and C both extend A
Multiple Inheritance (via class)	Not supported	Conflict issues
Multiple Inheritance (via interface)	Yes	class C implements A, B

Single Inheritance

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}
```

Multilevel Inheritance

```
class Animal {
    void eat() { System.out.println("Eats"); }
}

class Mammal extends Animal {
    void walk() { System.out.println("Walks"); }
}

class Dog extends Mammal {
    void bark() { System.out.println("Barks"); }
}
```

Hierarchical Inheritance

```
class Animal {
    void sound() { System.out.println("Some sound"); }
}

class Dog extends Animal {
    void bark() { System.out.println("Barks"); }
}

class Cat extends Animal {
    void meow() { System.out.println("Meows"); }
}
```

Why Java Does Not Support Multiple Inheritance (via classes)?

To avoid **ambiguity**, also known as the **diamond problem**.

The Role of super Keyword

super is used to:

1. **Call superclass constructor**
2. **Access superclass methods/variables that are hidden**

Example: Constructor chaining using super()

```
class Box {
    double width, height, depth;
    Box(double w, double h, double d) {
        width = w; height = h; depth = d;
    }
}

class BoxWeight extends Box {
    double weight;
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // calls Box constructor
        weight = m;
    }
}
```

Access hidden method/variable with super

```
class A {
    int x = 10;
    void show() { System.out.println("A’s show"); }
}

class B extends A {
    int x = 20;
    void show() {
        super.show(); // calls A’s show()
        System.out.println(super.x); // prints A’s x
    }
}
```

Object Slicing / Reference Limitation

A **superclass reference** can hold a **subclass object**, but only access superclass members:

```
Box obj = new BoxWeight(2, 3, 4, 5);
System.out.println(obj.width);   //  allowed
System.out.println(obj.weight);  //  compile error
```

Because the **type of the reference** (Box) decides what is accessible — **not** the type of object

Inheritance and Access Modifiers

- **private** members of superclass are not accessible directly in subclass.
- Use **protected** to allow access in subclass.
- public is accessible everywhere.

Constructor Execution Order

1. Superclass constructor runs **before** subclass constructor.
2. If you don't use super(), Java inserts default super() implicitly (no-arg constructor).
3. If superclass doesn't have a no-arg constructor, you **must** call super(args) explicitly.

Copy Constructor Using super

```
class Box {
    double w, h, d;
    Box(Box ob) {
        w = ob.w;
        h = ob.h;
        d = ob.d;
    }
}

class BoxWeight extends Box {
    double weight;

    BoxWeight(BoxWeight ob) {
        super(ob);       // calls Box(Box ob)
        weight = ob.weight;
    }
}
```

final Keyword in Inheritance

Use of final	Purpose
final class	Cannot be extended
final method	Cannot be overridden
final variable	Acts like a constant

Example

```
final class Vehicle {
    final void run() {
        System.out.println("Running...");
    }
}

// class Car extends Vehicle {}  Error: Cannot inherit final class
```

Notes on Inheritance

- Static methods are inherited, but they are **not overridden** — they are **hidden**.
- Static interface methods are **not inherited**.
- You **cannot override** a final method.
- super() must be the **first** statement in a constructor if used.
- You **can chain multiple constructors** using super(...).
- Inheritance allows **polymorphism**, but **not for variables** — only for methods.

Summary Table

Keyword	Purpose
extends	Enables class inheritance
super()	Calls parent class constructor
super.member	Accesses parent method/variable
final class	Prevents inheritance
final method	Prevents method override
private	Hides members from subclass
protected	Accessible to subclass

---

13. Polymorphism in Java

Polymorphism means **"many forms"**. In Java, it allows objects to behave differently based on their data type or class.
There are **two types** of Polymorphism:

- **Compile-time polymorphism** (Method Overloading → Early Binding)
- **Runtime polymorphism** (Method Overriding → Late Binding)

Method Overloading (Compile-Time Polymorphism)

What it means: You define **multiple methods with the same name** in a class, but with **different parameter types or counts**.

Java determines which method to call **at compile time**, based on **method signature** (parameters).

Example :
```
class OverloadDemo {

    void test(double a){
        System.out.println("Inside test(double) a: " + a);
    }
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;
        ob.test(i);        // converted int to double, calls test(double)
        ob.test(123.2);    // calls test(double)
    }
}
```

Explanation:

- Java sees test(int) doesn't exist.
- So it **automatically promotes** int to double.
- Hence, test(double) is called.

If test(int) had existed, Java would **prefer exact match** and call that instead.

Key Points:

- Overloading happens **within the same class**.
- **Return type doesn't matter** for overloading.
- Java uses **automatic type conversion** if no exact match is found.
- Resolved at **compile time** → This is **early binding**.

Method Overriding (Runtime Polymorphism)

What it means: If a **subclass provides a specific implementation** of a method already defined in its **superclass**, it is **overriding** the method.

Rules:

- Method name and parameter list must be **identical**.
- The return type can be **same or a subtype** (covariant return type).
- Access modifier of the subclass method must be **same or more visible**.

Example:
```
class A {

    void show() {
        System.out.println("Show from A");
    }
}
class B extends A {
    void show() {
        System.out.println("Show from B");
    }
}
public class Test {
    public static void main(String[] args) {
        A obj = new B();  // Superclass reference, subclass object
        obj.show();       // Calls B's show() → dynamic dispatch
    }
}
```

Output: `Show from B`

Key Points:

- Java decides at **runtime** which version of show() to execute.
- This is **dynamic method dispatch**, a core part of **runtime polymorphism**.
- Resolved at **runtime** → This is **late binding**.

Returning Objects (Object as Return Type)

```
class Test {
    int a;
    Test(int i) {
```

```
        a = i;
    }
    Test incrByTen() {
        Test temp = new Test(a + 10);
        return temp;
    }
}
```

Each time incrByTen() is called, a **new object is returned**, allowing method chaining or value retention.

This supports the **object-oriented principle** that everything is an object and can be passed or returned.

Early Binding vs Late Binding

Aspect	Early Binding	Late Binding
Also called	Static Binding	Dynamic Binding
Happens at	Compile Time	Runtime
Used in	Method Overloading	Method Overriding
Method resolution	Based on reference type	Based on actual object type
Performance	Faster	Slightly slower due to runtime check
Example	test(int), test(double)	show() in subclass B overrides class A

---

14. Encapsulation (Data Hiding)

Definition: Encapsulation is the process of **binding data (variables)** and **code (methods)** that operates on the data into a **single unit (class)** and **restricting direct access** to some of the object's components. This is usually done using **access modifiers (like private, public)**.

Goals of Encapsulation:

- Protect internal data from outside interference.
- Make code more **modular**, **maintainable**, and **secure**.
- Provide **controlled access** to data using **getters/setters**.

Real-Life Example: Think of a **bank account**. You can **deposit** or **withdraw** money, but you **can't directly access or modify** the balance in the database. That internal detail is hidden.

Java Example:
```
class BankAccount {

    private int balance; // 👈 Private field: cannot be accessed directly

    // Constructor
    public BankAccount(int initialBalance) {
        balance = initialBalance;
    }
```

```
    // Public method to access balance
    public int getBalance() {
        return balance;
    }

    // Public method to change balance
    public void deposit(int amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(int amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

Usage:

```
public class Main {

    public static void main(String[] args) {
        BankAccount acc = new BankAccount(1000);
        acc.deposit(500);
        acc.withdraw(200);
        System.out.println("Current Balance: " + acc.getBalance()); // 1300
    }
}
```

Key Points:

- balance is private → not accessible directly from outside.
- Methods deposit, withdraw, and getBalance provide controlled access.

---

15. Abstraction (Hiding Internal Implementation)

Definition: Abstraction means **hiding the complex implementation** details and **showing only the essential features** of an object.

You achieve abstraction in Java using:

- **Abstract classes**
- **Interfaces**

Real-Life Example: When you **drive a car**, you only interact with the **steering, brake, and accelerator** — you **don't care about** how the engine works internally. That internal engine mechanism is abstracted away.

Java Example using Abstract Class:

```
abstract class Animal {
    // Abstract method (no body)
```

```
    abstract void makeSound();

    // Non-abstract method
    void sleep() {
        System.out.println("Sleeping...");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark!");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow!");
    }
}
```

Usage:

```
public class Main {

    public static void main(String[] args) {
        Animal a = new Dog(); // Polymorphism
        a.makeSound();  // Bark!
        a.sleep();      // Sleeping...
    }
}
```

Java Example using Interface:

```
interface Vehicle {
    void start();
    void stop();
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car started");
    }

    public void stop() {
        System.out.println("Car stopped");
    }
}
```

Usage:

```
public class Main {

    public static void main(String[] args) {
        Vehicle v = new Car();
        v.start();  // Car started
        v.stop();   // Car stopped
    }
}
```

Summary Table

Concept	Encapsulation	Abstraction
What	Hides data (variables)	Hides implementation details
How	Using private fields + getters/setters	Using abstract class or interface
Access	Restricted to methods only	Focus on what the object **does**, not how
Goal	Data protection and control	Reduce complexity, improve usability
Real-life	Bank Account (can't access balance directly)	Car (you drive without knowing engine details)

When to Use

- Use **encapsulation** to protect sensitive data and provide controlled access.
- Use **abstraction** when you want to **simplify a complex system**, focusing only on high-level behavior.

Abstraction	Encapsulation
Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data into a **single** entity or unit so that the data can be protected from the outside world.
It solves an issue at the **design** level.	Encapsulation solves an issue at **implementation** level.
It focuses on the **external** lookout.	It focuses on **internal** working.
It can be implemented using **abstract classes** and **interfaces**.	It can be implemented by using the **access modifiers** (private, public, protected).
It is the process of **gaining** information.	It is the process of **containing** the information.
In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities.	We use the **getters** and **setters** methods to hide the data.
The objects are **encapsulated** that helps to perform abstraction.	The object need not to **abstract** that result in encapsulation.

---

16. What is an Abstract Class?

An **abstract class** is a class that **cannot be instantiated** (i.e., you cannot create objects from it), but it can be **subclassed**. It provides a **template or blueprint** for other classes to extend and fill in the details.

Why Abstract Class?

Sometimes, you want a **superclass** to define a **common structure**, but **not the full implementation**. So, you make the class **abstract**, and force the subclasses to implement the **incomplete methods**.

Real-Life Analogy Think of a **"Vehicle"** class:

- All vehicles have **start()**, **stop()**, **fuelType()**
- But **how** they implement start() or fuelType() depends on whether it's a car, bike, or electric scooter

So you can define Vehicle as an **abstract class** and leave method fuelType() **abstract** (unimplemented) so each subclass must define it.

Key Concepts:

Abstract Method:

```
abstract return_type methodName(parameters);
```

- No body (implementation).
- Must be overridden in subclass unless subclass is also abstract.

Abstract Class:

- A class with **at least one abstract method**.
- Cannot be instantiated.
- Can have:
    - Constructors (yes!)
    - Fields (static, non-static, final, etc.)
    - Concrete methods (with implementation)
    - Static methods

Java Code Example

```
// Abstract class
abstract class Vehicle {
    int speed = 0;

    Vehicle() {
        System.out.println("Vehicle created");
    }

    void start() {
        System.out.println("Vehicle starting...");
    }

    abstract void fuelType();  // Must be overridden
}

// Subclass
class Car extends Vehicle {
    @Override
    void fuelType() {
```

```
        System.out.println("Car uses petrol or diesel");
    }
}

public class Main {
    public static void main(String[] args) {
        // Vehicle v = new Vehicle(); //  Not allowed
        Vehicle v = new Car();       //  Allowed: reference of abstract
class
        v.start();                   // Concrete method from abstract class
        v.fuelType();                // Overridden abstract method
    }
}
```

Output:

```
Vehicle created
Vehicle starting...
Car uses petrol or diesel
```

Rules & Facts (Important):

Rule	Explanation
You **cannot create objects** of abstract class	Because it's incomplete
Abstract class **can have constructor**	Used during subclass object creation
You **cannot have abstract constructors or static abstract methods**	Because they don't make sense (constructors are for object creation, which abstract class doesn't allow)
A subclass must **override all abstract methods**, or itself be marked abstract	
You can have **non-abstract (concrete) methods** in abstract class	
Abstract class can **have static methods**	But **not abstract static** methods

Abstract Class vs Interface

Feature	Abstract Class	Interface
Methods	Abstract & Concrete	Abstract only (Java 8+ allows default & static)
Constructors	Yes	❌ No
Variables	final, non-final, static, non-static	All variables are public static final
Access Modifiers	private, protected, public	Only public
Inheritance	One abstract class (single inheritance)	Multiple interfaces
Purpose	Partial abstraction (0–100%)	Full abstraction (100%)

When to use what?

Use Abstract Class when	Use Interface when
You want to share code	You want to ensure a contract
You expect future evolution	You want multiple inheritance
You want controlled accessibility (private/protected)	All methods public, no state logic needed

Summary

- Use **abstract classes** when you want to define a base class with some **shared behavior** and some **must-override methods**.
- Abstract classes are best for when you want a class to define a **general structure** but leave some **details to be filled in** by subclasses.
- They support **inheritance**, **encapsulation**, and **runtime polymorphism**.
- Cannot be instantiated directly.

---

17. What Are Access Modifiers?

In **Java**, **access modifiers** are **keywords** that set the **visibility (or access level)** of **classes**, **methods**, and **variables** to control how they are **accessed across other classes and packages**.

They are key tools for **encapsulation**, **security**, and **modular code design**.

The Four Access Modifiers in Java

Access Modifier	Access Level	Keyword
public	Everywhere	public
protected	Subclass + Package	protected
(default)	Same Package Only	*(no keyword)*
private	Only Inside the Class	private

Access Modifier Table

Modifier	Same Class	Same Package	Subclass (diff pkg)	World (anywhere)
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

Real-Life Analogy Imagine a company **with:**

- **Private**: Only the employee (class) knows their salary (field).

- **Default**: Shared documents within one department (package).
- **Protected**: Training procedures shared with team and new hires (subclasses).
- **Public**: Company website (methods or classes available to everyone).

Modifier-by-Modifier Breakdown

1. public — Open to All

- Accessible from **anywhere** in any package.
- Common for utility classes or API methods.

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

2. private — Fully Hidden

- Accessible **only** inside the class where it is declared.
- Often used for **data hiding** and **encapsulation**.

```
public class Account {
    private double balance;

    public void deposit(double amt) {
        balance += amt;
    }

    private void audit() {
        // Only class methods can call this
    }
}
```

3. default (package-private)

- When **no modifier** is used.
- Accessible only **within the same package**.
- Good for **package-level cohesion**, not visible outside.

```
class Logger {
    void logInfo(String msg) {
        System.out.println(msg);
    }
}
```

4. protected — Accessible in Package + Subclasses

- Accessible in:

- Same package
- Subclasses in other packages *(with object caveat explained below)*

Important Protected Rule:

You can access a **protected** member of a superclass **from a subclass in a different package only if you're calling it on this or a subclass object**, **not a superclass object**.

Not allowed:

```
Base base = new Base();
base.display();  // ✕ not allowed if you're in a different package
```

Allowed:

```
Derived d = new Derived();
d.display();     // ✓ allowed (you're using subclass reference)
display();       // ✓ also allowed within the subclass
```

Even though Derived is a subclass, it can't access protected members **via superclass reference**.

Full Example:

packageOne/Base.java

```
package packageOne;

public class Base {
    protected void display() {
        System.out.println("In Base");
    }
}
```

packageTwo/Derived.java

```
package packageTwo;
import packageOne.Base;

public class Derived extends Base {
    public void show() {
        display(); // ✓ OK: accessing inherited protected method
        new Derived().display(); // ✓ OK: subclass object
        new Base().display();    // ✕ ERROR: not same package, not
subclass object
    }
}
```

Why Java is strict with protected?

Java restricts access to protected members via **parent-type references from outside packages** to prevent **unexpected interference across unrelated subclasses**.

This ensures:

- Subclasses **only manipulate their own protected fields**, or those of their **own type**
- Prevents classes from modifying internals of **other subclasses** they shouldn't even know about

Best Practices

Scenario	Recommended Modifier
Internal logic only	private
Within team of classes in same package	*(default)*
Extendable utility code	protected
Public APIs / libraries	public

Summary

- **Access modifiers control visibility** across classes and packages.
- private ⟶ strictest, public ⟶ most open.
- **Use protected carefully**: It's not the same as "subclass can access everything".
- Access rules depend on:
  - Where the class is
  - Whether it's a subclass
  - Whether you're using a superclass or subclass reference

---

18. What is an Interface in Java?

An **interface** in Java is a **reference type**, similar to a class, **but with no method bodies** (until Java 8+). It defines a **contract** (what a class must do), **not how it does it**.

Think of an interface as a **"pure abstraction"**: it tells what methods a class must implement, without saying how.

Why Interfaces?

Java does **not support multiple inheritance** through classes to avoid the **diamond problem** (ambiguity when two superclasses have same method).

Instead, Java uses **interfaces** to allow a class to **inherit behavior from multiple sources** safely.

Interface Syntax

```
interface Vehicle {
    void start();          // implicitly public & abstract
    int MAX_SPEED = 120;  // implicitly public, static, final
}
```

All methods in an interface are **public and abstract** by default (until Java 8+).
All variables are **public, static, final** (constants) and **must be initialized**.

Implementing an Interface

```
class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting...");
    }
}
```

implements is used instead of extends to apply an interface to a class.
All interface methods **must be implemented** unless the class is abstract.

Multiple Interfaces = Multiple Inheritance

```
interface A {
    void show();
}
interface B {
    void display();
}

class Test implements A, B {
    public void show() { System.out.println("show"); }
    public void display() { System.out.println("display"); }
}
```

One class can implement **multiple interfaces** (which is not possible with multiple classes).

Interface vs Abstract Class

Feature	Interface	Abstract Class
Inheritance	Can be implemented by many classes	Can be extended by only one class
Multiple Inheritance	✅ Yes	❌ No
Constructors	❌ Not allowed	✅ Can have
Variables	public static final only	Any type, including instance fields
Methods	abstract, default, static	abstract and concrete
State (fields)	Cannot store object state	Can maintain state

Interface Features: In-Depth

Default Methods (Java 8+)

You can define methods **with a body** in an interface using default.

```
        interface Printer {
    default void print() {
        System.out.println("Default printing...");
    }
}
```

Helps to **add new methods** to interfaces **without breaking** old implementations.

If two interfaces have **conflicting default methods**, the class **must override** it, or a **compile-time error** occurs

Static Methods in Interfaces

```
interface Tool {
    static void info() {
        System.out.println("Static method in interface");
    }
}
```

Must have a body. Not inherited by classes. Call using Tool.info().

Nested Interfaces

An interface can be declared **inside a class or another interface**.

```
class Outer {
    interface Nested {
        boolean isValid(int x);
    }
}

class Impl implements Outer.Nested {
    public boolean isValid(int x) {
        return x > 0;
    }
}
```

A nested interface can be public, private, or protected.

Polymorphism with Interfaces

You can reference objects using an **interface type**, similar to superclass reference.

```
Vehicle v = new Car(); // Polymorphic reference
v.start();             // Calls Car's start at runtime
```

Method resolution is done **dynamically at runtime**, based on the object **being referenced**.

Why this matters:

Interfaces provide **flexibility** and **decoupling**:

- Code that **uses the interface** doesn't need to know the specific class.
- Encourages **clean architecture**, **extensibility**, and **testability**.

Pitfall: No Instance Variables

Unlike classes or abstract classes:

```
interface MyInterface {
    int x = 10;  // public static final by default
}
```

You **cannot** do:

```
x = 20; // ERROR: cannot assign a value to final variable
```

Method Access and Signatures

When implementing interface methods:

- Must be public
- Must match the **signature exactly** (return type, name, params)

```
interface Animal {
    void makeSound();
}

class Dog implements Animal {
    public void makeSound() {} // ✅
    void makeSound() {}        // ❌ not public
}
```

Example: Real Use Case

```
interface Drawable {
    void draw();
}
interface Scalable {
    void resize();
}

class Circle implements Drawable, Scalable {
    public void draw() {
        System.out.println("Drawing Circle");
    }
    public void resize() {
        System.out.println("Resizing Circle");
    }
}
```

Circle inherits behavior contracts from **two sources**, fulfilling both.

Summary: Key Points to Remember

Topic	Summary
Can store state?	No instance variables
Method types	abstract, default, static
Variable type	public, static, final
Multiple inheritance	Yes via interface
Implements	One class can implement many interfaces
Syntax	interface, implements, default, static
Use case	Defining capability contracts, decoupling logic
Dynamic dispatch	Supported via interface references

---

19. Generics in Java

What are Generics? : Generics allow you to create classes, interfaces, and methods where the **type of data** is **specified as a parameter**.

Think of generics as a way to write code that works with **any type** without losing **type safety**.

Why Use Generics?

- **Type safety**: Catch errors at **compile time**.
- **Code reusability**: Write methods/classes that work with any type.
- **Avoid casting**: No need to cast from Object.

Syntax:
```
class Box<T> {

    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```

Usage Example:

```
Box<Integer> intBox = new Box<>();

intBox.set(123);
System.out.println(intBox.get());   // Output: 123
```

```
Box<String> strBox = new Box<>();
strBox.set("Aadii");
System.out.println(strBox.get());  // Output: Aadii
```

Generic Method:

```
public <T> void printArray(T[] arr) {
    for (T item : arr) {
        System.out.println(item);
    }
}
```

Common Mistakes:

- You **can't use primitives** like int, double. Use wrapper classes: Integer, Double, etc.
- You **can't create instances** of generic type: T obj = new T(); // ❌ not allowed

Bounded Generics:

```
public <T extends Number> void printDouble(T num) {
    System.out.println(num.doubleValue());
}
```

---

20. Comparable Interface

Purpose: Used for **natural ordering** of objects, like sorting students by marks, strings alphabetically, etc.

Syntax: `class Student implements Comparable<Student> {`

```
    int marks;
    String name;

    public Student(int marks, String name) {
        this.marks = marks;
        this.name = name;
    }

    @Override
    public int compareTo(Student other) {
        return this.marks - other.marks; // Ascending order
    }
}
```

Sorting Example:

```
List<Student> students = new ArrayList<>();
students.add(new Student(70, "A"));
students.add(new Student(90, "B"));

Collections.sort(students);  // Uses compareTo
```

compareTo return values:

- < 0: current object < other
- 0: current object == other
- > 0: current object > other

Common Mistakes:

- Forgetting to implement compareTo() causes runtime errors when sorting.
- Confusing Comparable with Comparator (Comparator is external/custom order).

---

21. Lambda Functions (Java 8+)

What is a Lambda? A **lambda expression** is a **short way to write anonymous methods** (functions without names).

Used mainly to:

- Replace **anonymous classes**
- Simplify **functional interfaces** (interfaces with one abstract method)

Syntax: `(parameters) -> expression`

OR `(parameters) -> {`

```
    // multiple statements
}
```

Example: Runnable

```
Runnable r = () -> System.out.println("Hello from thread");
new Thread(r).start();
```

Example: Sorting with Lambda

```
List<String> names = Arrays.asList("Adnan", "Jyoti", "Rahul");

Collections.sort(names, (a, b) -> a.compareTo(b)); // Lambda replaces
Comparator
```

Functional Interface Example:

```
@FunctionalInterface
```

```
interface Greeting {
    void sayHello(String name);
}

public class Test {
    public static void main(String[] args) {
        Greeting g = (name) -> System.out.println("Hello " + name);
        g.sayHello("Aadii");
    }
}
```

Real Use Case: forEach

```
List<Integer> nums = Arrays.asList(1, 2, 3);
nums.forEach(n -> System.out.println(n));
```

Common Mistakes:

- Using lambdas where multiple abstract methods exist — **won't work** unless the interface is functional.
- Misunderstanding that lambdas don't capture variables like regular closures — they **can** capture **effectively final** variables.

Summary Table

Feature	Use	Key Benefit
Generics	Type-safe reusable code	Avoids casting and runtime errors
Comparable	Natural object ordering	Enables sorting with Collections
Lambda	Replace anonymous classes	Shorter, cleaner code

---

22. What is Exception Handling?

Exception handling is a mechanism to handle **runtime errors** so that normal flow of the application can be maintained.

Real-life analogy: Think of exception handling like a **seatbelt in a car** – it doesn't prevent the accident (error), but it helps **reduce the damage** (crash of your program).

Java Exception Hierarchy All exceptions/errors in Java inherit from:

```
java.lang.Throwable
├── Error       (serious issues like JVM crash, OutOfMemory)
└── Exception   (things we can handle)
     ├── RuntimeException (unchecked)
     └── IOException, SQLException, etc. (checked)
```

Types of Exceptions

Type	Examples	Must Handle?
Checked	IOException, SQLException	YES – must handle using try/catch or throws
Unchecked	NullPointerException, ArrayIndexOutOfBoundsException	NO – optional

Keywords in Exception Handling

Keyword	Meaning
try	Code that might throw an exception
catch	Handles the exception
finally	Always runs (whether exception is thrown or not)
throw	Manually throw an exception
throws	Declare an exception in method signature

Basic Syntax:

```
try {
    // risky code
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Can't divide by zero!");
} finally {
    System.out.println("This will always run.");
}
```

Custom Exception (User-defined)

You can create your own exceptions by extending Exception class.

```
class AgeTooSmallException extends Exception {
    public AgeTooSmallException(String message) {
        super(message);
    }
}

public class Test {
    static void checkAge(int age) throws AgeTooSmallException {
        if(age < 18) {
            throw new AgeTooSmallException("Age must be 18+");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (AgeTooSmallException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Throwable Class in Java

It's the superclass of all errors and exceptions.

```
public class Throwable implements Serializable {
    private String detailMessage;
    private Throwable cause;
}
```

Common Methods:

- getMessage() → returns the message of exception.
- printStackTrace() → prints where the exception occurred.
- getCause() → returns the cause of the exception.
- toString() → returns string representation.

Exception Propagation

If a method doesn’t catch an exception, it goes up the call stack.

```
public class Test {
    static void m1() {
        int a = 10 / 0;
    }

    static void m2() {
        m1(); // no try-catch
    }

    public static void main(String[] args) {
        m2(); // Exception will be thrown here
    }
}
```

You can use throws to pass exception up:

```
void readFile() throws IOException {
    FileReader fr = new FileReader("file.txt");
}
```

Nested Try-Catch

```
try {
    try {
        int[] arr = new int[5];
        arr[10] = 50;
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Inner catch");
    }
} catch (Exception e) {
    System.out.println("Outer catch");
}
```

try-with-resources Example

```
try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}
```

No need to close the file manually!

Error vs Exception vs Throwable

Type	Can Catch?	Example	Meaning
Throwable	Yes	Both Error and Exception	Root of all problems
Exception	Yes	IOException, SQLException	Recoverable
Error	No	OutOfMemoryError, StackOverflowError	Unrecoverable

Summary Table

Concept	Meaning
Throwable	Root of all errors and exceptions
Exception	Recoverable problems (file not found, wrong input)
Error	Serious problems (JVM crash, memory overflow)
Checked Exception	Must be handled (compile-time)
Unchecked Exception	Runtime exception (can ignore)
try-catch-finally	Used to handle and clean up
throw	Used to manually throw exception
throws	Declare the exception from method

---

23. What is Object Cloning?

In Java, **object cloning** is the process of **creating an exact copy of an object**.

Java provides a mechanism to clone objects through the Cloneable interface and the clone() method in the Object class.

Why use cloning?

- To **duplicate complex objects** without manually copying each field.
- Useful in **caching**, **prototyping**, and **undo operations**.

How does it work?

Java's default cloning is **shallow**. To use it:

1. Your class must **implement the `Cloneable` interface**.
2. Override the `clone()` method.
3. Call `super.clone()` inside your `clone()` method.

Example:

```
class Student implements Cloneable {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone(); // Shallow copy
    }
}

public class Main {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Student s1 = new Student(1, "Aadii");
        Student s2 = (Student) s1.clone();

        System.out.println(s1.name); // Aadii
        System.out.println(s2.name); // Aadii
    }
}
```

What is Shallow Copy?

A **shallow copy** copies **only the reference** of objects (not actual objects).

This means:

- Primitive fields are copied as values.
- Object fields are **shared** between original and cloned objects.

Example of Shallow Copy:

```
class Address {
    String city;
    Address(String city) { this.city = city; }
}

class Employee implements Cloneable {
    String name;
    Address address;

    Employee(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone(); // shallow copy
    }
}
```

```java
public class Main {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Address addr = new Address("Indore");
        Employee e1 = new Employee("Aadii", addr);
        Employee e2 = (Employee) e1.clone();

        e2.address.city = "Bhopal";

        System.out.println(e1.address.city); // Bhopal (affected!)
        System.out.println(e2.address.city); // Bhopal
    }
}
```

As you can see, both e1 and e2 point to the **same Address object**.

What is Deep Copy?

A **deep copy** copies the object and **all objects inside it recursively**. So both the original and clone are completely independent.

How to do Deep Copy in Java?

Manually clone the nested objects:

```java
class Address {
    String city;
    Address(String city) { this.city = city; }

    Address(Address addr) {
        this.city = addr.city;
    }
}

class Employee {
    String name;
    Address address;

    Employee(String name, Address address) {
        this.name = name;
        this.address = new Address(address); // deep copy
    }
}
```

Now both the original and copy have separate `Address` objects.

Difference Summary

Feature	Shallow Copy	Deep Copy
Object Cloning	References copied	Entire object tree copied
Performance	Faster	Slower (recursive copying)
Independence	Shared object references	Fully independent objects
Use clone()	Yes (default behavior)	Yes (with custom clone logic)

24. What is the Java Collections Framework?

The **Java Collections Framework (JCF)** is a **set of classes and interfaces** in Java that provide **standardized ways to store, access, and manipulate groups of objects** (like arrays, lists, maps, sets, etc.).

Think of it like a **toolbox** of ready-made data structures and utilities for handling groups of data efficiently.

Why Use Collections?

- Dynamic sizing (no need to specify fixed size like arrays)
- Easier searching, sorting, and manipulation
- Built-in methods (like .add(), .remove(), .sort(), .contains(), etc.)
- Saves time and improves performance

Core Interfaces of Java Collections

All collection classes are based on a few key **interfaces**:

Interface	Description
Collection	Root interface of the collection hierarchy
List	Ordered collection, can contain duplicates
Set	Unordered collection, **no duplicates**
Queue	Follows FIFO (First In, First Out)
Deque	Double-ended queue (insert/remove from both ends)
Map	Stores key-value pairs (not part of Collection interface directly)

Interface Hierarchy Diagram:

```
                |
             Collection
        _____________|_____________
       |             |              |
     List           Set            Queue
       |             |              |
   ArrayList     HashSet        PriorityQueue
   LinkedList    LinkedHashSet  ArrayDeque
   Vector        TreeSet

                    Map
             __________|__________
            |          |           |
   HashMap    LinkedHashMap   TreeMap
```

Detailed Look at Key Interfaces and Classes

List – Ordered, allows duplicates

- Maintains insertion order.
- Access by index (like an array).
- Allows null values.

Implementations:

ArrayList

- **Resizes dynamically**
- Fast random access (O(1))
- Slower insertion/deletion in middle

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
System.out.println(list.get(1)); // Banana
```

LinkedList

- Doubly-linked list
- Fast insertion/deletion (O(1) at ends)
- Slower random access

```
LinkedList<Integer> ll = new LinkedList<>();
ll.addFirst(10);
ll.addLast(20);
System.out.println(ll); // [10, 20]
```

Vector (legacy)

- Thread-safe but slower
- Rarely used in new code

Set – No duplicates allowed

HashSet

- Unordered
- Uses hash table
- Allows one null

```
Set<Integer> set = new HashSet<>();
set.add(10);
set.add(20);
set.add(10); // Duplicate, won't be added
```

LinkedHashSet

- Maintains insertion order
- Slower than HashSet but predictable iteration

```
Set<String> lhs = new LinkedHashSet<>();
lhs.add("A");
lhs.add("B");
System.out.println(lhs); // [A, B]
```

TreeSet

- Sorted set (natural or custom comparator)
- No nulls allowed
- Internally uses a Red-Black Tree

```
Set<Integer> ts = new TreeSet<>();
ts.add(3);
ts.add(1);
ts.add(2);
System.out.println(ts); // [1, 2, 3]
```

Queue / Deque

PriorityQueue

- Elements ordered based on priority
- Min-heap by default

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(30);
pq.add(10);
pq.add(20);
System.out.println(pq.poll()); // 10
```

ArrayDeque

- Fast stack and queue operations
- No nulls allowed

```
Deque<String> deque = new ArrayDeque<>();
deque.addFirst("Front");
deque.addLast("Back");
System.out.println(deque); // [Front, Back]
```

Map – Key-value pairs (not part of Collection interface)

HashMap

- Unordered
- Allows one null key and multiple null values
- Fast lookups

```
Map<String, Integer> map = new HashMap<>();
map.put("Aadii", 100);
map.put("GPT", 90);
```

```
System.out.println(map.get("Aadii")); // 100
```

LinkedHashMap

- Maintains insertion order
- Good for LRU cache

TreeMap

- Sorted by keys
- No null keys

```
Map<Integer, String> tm = new TreeMap<>();
tm.put(2, "B");
tm.put(1, "A");
System.out.println(tm); // {1=A, 2=B}
```

Utility Class: Collections

Java provides a utility class java.util.Collections with static methods:

```
Collections.sort(list);
Collections.reverse(list);
Collections.shuffle(list);
Collections.max(list);
Collections.min(list);
```

Differences Summary

Feature	List	Set	Map
Duplicates	Yes	No	Keys no, values yes
Order	Maintained (List, LinkedHashSet)	Not always	LinkedHashMap maintains
Nulls	Yes	Yes (depends)	HashMap allows one null key

Real-Life Example

Suppose you're building an **online shopping cart**:

- Use ArrayList to store items in the order added.
- Use HashSet to keep track of unique user IDs.
- Use HashMap to store item name and quantity or price.
- Use PriorityQueue to process orders based on urgency.
- Use LinkedHashMap for recently viewed items (LRU cache).

- Know time complexities:
  `ArrayList` – O(1) access, O(n) insert
  `LinkedList` – O(1) insert/remove, O(n) access

`HashSet/HashMap` – O(1) operations (average case)
`TreeMap/TreeSet` – O(log n)

- Be ready to write comparators for `TreeSet` or `PriorityQueue`

---

25. Vector Class in Java

◆ What is a Vector?

- Vector is a **legacy class** (from Java 1.0) and part of the **java.util package**.
- It is a **resizable array** (just like ArrayList) but **synchronized** — meaning it's **thread-safe**.

Key Features:

Feature	Description
Dynamic Array	Grows/shrinks as needed
Synchronized	Thread-safe by default
Can contain duplicates	Yes
Maintains insertion order	Yes
Nulls allowed?	Yes
Implements?	List, RandomAccess, Cloneable, Serializable

Constructors:

```
Vector()                      // Default initial capacity is 10
Vector(int initialCapacity)
Vector(int initialCapacity, int capacityIncrement)
Vector(Collection<? extends E> c)
```

Important Methods:

```
add(E e)               // Add at end
add(int index, E e)    // Add at position
get(int index)         // Get element
remove(int index)      // Remove element
size()                 // Current number of elements
capacity()             // Current capacity
setSize(int newSize)   // Resize vector
ensureCapacity(int minCapacity)   // Ensure it has at least minCapacity
```

Example:

```
import java.util.Vector;

public class VectorExample {
    public static void main(String[] args) {
        Vector<String> vec = new Vector<>();
```

```
        vec.add("A");
        vec.add("B");
        vec.add("C");
        vec.add("D");

        System.out.println("Vector: " + vec);
        System.out.println("Element at index 2: " + vec.get(2));
        System.out.println("Size: " + vec.size());
        System.out.println("Capacity: " + vec.capacity());
    }
}
```

Vector vs ArrayList:

Feature	Vector		ArrayList
Thread-safe	Yes		No (non-sync)
Performance	Slower (synchronization overhead)		Faster
Introduced in	Java 1.0		Java 1.2

---

26. Enums in Java (Enumeration)

What is an Enum?

- Enum is a **special class type** that defines a fixed set of **named constants**.
- Declared using the `enum` keyword.
- Unlike C/C++, Java enums are **much more powerful** — they are **classes in disguise**.

Basic Example:

```
enum Color {
    RED, BLUE, GREEN
}
```

Internally, Java converts this enum to:

```
class Color {
    public static final Color RED = new Color();
    public static final Color BLUE = new Color();
    public static final Color GREEN = new Color();
}
```

Enums are Classes:

- You can add:
    - Constructors
    - Fields
    - Methods
    - Implement interfaces

But: **cannot extend a class** (because enum already extends java.lang.Enum implicitly).

Full Example with Constructor and Method:

```
enum Size {
    SMALL(30), MEDIUM(40), LARGE(50);

    private int radius;

    // Constructor (must be private or package-private)
    Size(int radius) {
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }
}

public class EnumTest {
    public static void main(String[] args) {
        Size s = Size.MEDIUM;
        System.out.println("Size: " + s);
        System.out.println("Radius: " + s.getRadius());
    }
}
```

Enum Methods from java.lang.Enum:

Method	Description
values()	Returns all enum constants as an array
ordinal()	Returns the position (index) of enum constant
valueOf()	Returns enum constant from a string name
toString()	Returns name of the constant (can override)

Example:

```
for (Size s : Size.values()) {
    System.out.println(s + " has ordinal " + s.ordinal());
}

Size s = Size.valueOf("LARGE");  // From string
```

Enum Comparison:

```
Size s1 = Size.SMALL;
Size s2 = Size.SMALL;

System.out.println(s1 == s2);          // true
System.out.println(s1.equals(s2));     // true
```

- **Use ==** or .equals() — both are safe and compare actual object reference.
- Enum constants are **singleton**.

Enums Can Implement Interfaces:

```
interface Printable {
    void print();
}

enum Status implements Printable {
    SUCCESS, FAILURE;

    public void print() {
        System.out.println("Status: " + this);
    }
}
```

main() method inside Enum:

```
enum Direction {
    EAST, WEST, NORTH, SOUTH;

    public static void main(String[] args) {
        System.out.println("Enum main method: " + Direction.NORTH);
    }
}
```

Yes, you can run an enum directly via command prompt using java Direction.

Enum Limitations:

- Can't extend other classes (already extends Enum)
- Constructors **must be private/package-private**
- Can't create enum objects explicitly (new is not allowed)
- Can't define abstract methods

Real-Life Use Cases:

- Days of the week
- Status codes (SUCCESS, FAILURE)
- States in a process (STARTED, RUNNING, STOPPED)
- HTTP methods (GET, POST, DELETE...)

Best Practices:

Use enums instead of int or String constants
Add constructors and methods if needed
Use values(), valueOf(), and ordinal() smartly
Use switch-case with enums for better readability

Bonus: Enum with Switch

```
enum Day {
    MON, TUE, WED, THU, FRI, SAT, SUN
}
```

```java
public class TestEnum {
    public static void main(String[] args) {
        Day today = Day.FRI;
        switch(today) {
            case FRI:
                System.out.println("Weekend is near!");
                break;
            default:
                System.out.println("It's a weekday.");
        }
    }
}
```

Summary

Feature	Vector	Enum
Type	Class (legacy)	Special class (enum)
Thread-safe	Yes	Not related
Purpose	Dynamic array	Fixed set of named constants
Key Concepts	Synchronized, resizable	Singleton constants, type-safe
Uses	When thread safety is needed	To replace constants, states

---

MADE WITH ❤ BY ADNAN QURESHI